UNIVERSITÉ DE MONTRÉAL

CENTRE DE RECHERCHE SUR LES TRANSPORTS

CASE POSTALE 6128, SUCCURSALE A, MONTRÉAL, QUÉBEC, H3C 3J7

ALTO:
A COMPUTER SYSTEM FOR THE DESIGN AND EXPERIMENTATION OF ROUTING ALGORITHMS

by

Jean-Yves Potvin*
Guy Lapalme*
Jean-Marc Rousseau*+

* Département d'Informatique et de Recherche opérationnelle, Université de Montréal.

+ Centre de recherche sur les transports, Université de Montréal.

Cette étude a été publiée grâce à une subvention du fonds FCAR pour l'aide et le soutien à la recherche.

Université de Montréal

Centre de recherche sur les transports - Publication #525
Département d'informatique et de recherche opérationnelle - Publication #616

June 1987

Dépôt légal - 2e trimestre 1987

Abstract. This paper describes the computer system ALTO, designed to facilitate the development and experimentation of routing algorithms for transportation vehicles. This system, implemented on the 1108 Xerox Lisp Machine, embodies a set of templates, known as the "general heuristic", which is instantiated by an expert user with his own formulae. In this way, ALTO can be used to reproduce a broad class of classical routing algorithms already described in the litterature or can be used to devise new resolution strategies in face of complex problems. A real application for a mail pick-up problem is presented at the end of the paper, in order to emphasize the benefits of the system.

Key Words. Computer System, General Heuristic, Object-Oriented Programming, Routing.

Résumé. Le présent article introduit le système informatique ALTO conçu afin de favoriser le développement et l'expérimentation d'algorithmes de génération de tournées pour des véhicules de transport. Ce système, implanté sur la machine Lisp Xerox 1108, repose principalement sur une heuristique générale, soit un ensemble de modèles susceptibles d'être instanciés par un utilisateur expert à l'aide de ses propres formules. Il devient dès lors possible de reproduire de nombreux algorithmes classiques décrits dans la littérature ou encore de concevoir de nouvelles approches de résolution face aux problèmes rencontrés. Une application concrète dans le domaine de la collecte du courrier est d'ailleurs présentée à la fin de l'article afin de souligner l'intérêt d'un tel système.

Mots Clés. Système Informatique, Heuristique Générale, Programmation orientée objet, Tournées.

ALTO:
A Computer System for the Design and Experimentation of Routing Algorithms

Introduction

The aim of this paper is to describe ALTO, an interactive and graphic system designed to facilitate the development and experimentation of routing algorithms for transportation vehicles. Using ALTO, the designer of algorithms can define and test quickly a multiplicity of new (or already known) heuristic strategies in face of problems for which no exact polynomially bounded algorithms exist. This flexibility is a major asset since the identification of good resolution strategies for complex and constrained problems is rarely obvious and is closely linked to the characteristics of the problems. The experimentation work done with our system will hopefully enhance the development of the expertise in the field of vehicle routing and lead to an expert system able to suggest automatically good heuristic strategies.

The system ALTO is written in LOOPS, an object-oriented language, and is implemented on the 1108 Xerox Lisp Machine which provides a rich and powerful environment known as INTERLISP-D [11, 12]. Since the problem solving process is based on a general heuristic instantiated with user's formulae in order to guide the system toward solutions, the interactive and graphic features of the INTERLISP-D environment, like menus and windows, are greatly sollicitated. In fact, this environment allows a direct intervention of the user, even during the solving process, because it is now possible to visualize the routes produced by any instantiation formulae and to revise them if the result appears unsatisfactory.

Hence, the major features of the system are:

a) The graphic features for the display and construction of the network.

ALTO can display networks on the screen, using the window system offered by the INTERLISP-D environment. It is then possible to modify dynamically the structure of these networks by moving the nodes to new locations or by adding and deleting nodes and links. Naturally, the display is automatically updated when these modifications take place.

Besides the editing functions, many other graphic features are available in the system, like the modification of the scale of the network, the display of the routes serviced by the vehicles, etc. These facilities are very useful when the user wants to examine in detail the routes produced by the general heuristic (figure 1).

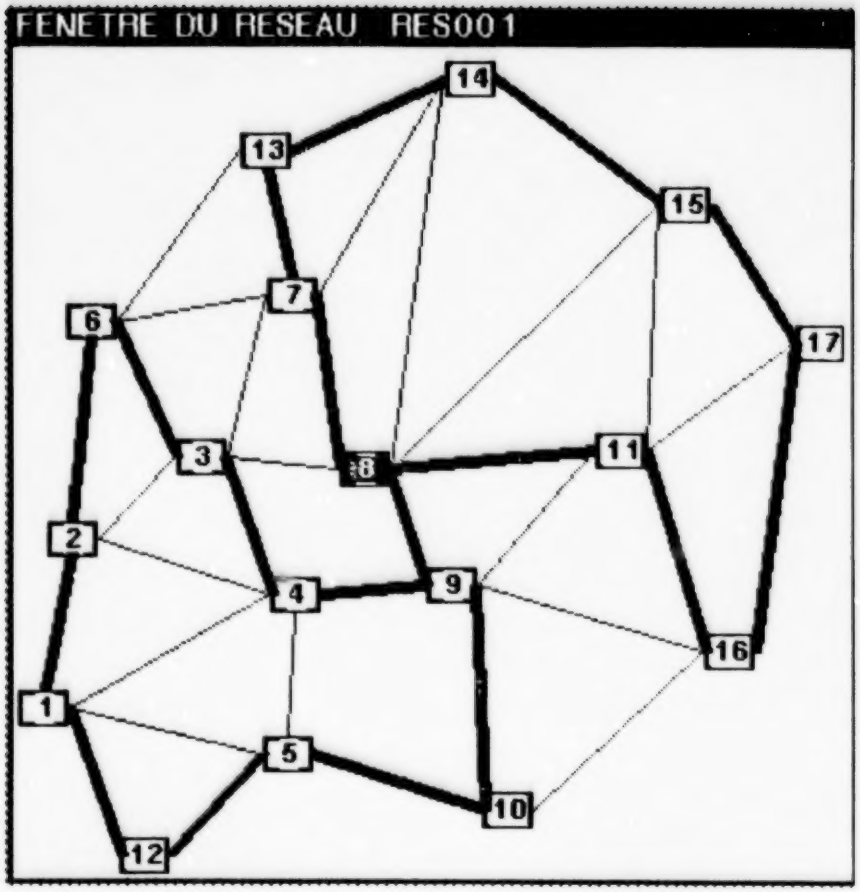

Fig. 1 Routes for two vehicles

b) The interactive features for the specification of the problem.

With the menus offered, the user is able to create new problems interactively. For instance, it is possible to add or remove vehicles, to redefine the set of stops or clients serviced by the vehicles, to relocalize the origin (depot) and to modify the major characteristics of the problem like the capacity of the vehicles or the demand to be collected at the stops.

c) The algorithmic features for the resolution of the problem

Algorithmic facilities are embedded in the system in order to allow the resolution of the problem. These facilities include for now a Dijkstra's algorithm, an exact Traveling Salesman's algorithm and the general heuristic which will be fully described in sections 3 to 9. Other algorithms like a general assignment algorithm and a matching algorithm will eventually be added.

Section 1. General concepts

ALTO is actually designed to solve a class of transportation problems known as "node routing problems" [2]. In these problems, the demand is located on the nodes of the network as opposed to "arc routing problems" where the demand is located on the links (a new version of ALTO for arc routing problems is planned). Since various node routing problems are described in the litterature, each with its own peculiarities, we restricted somewhat the field of application of the system in the current implementation and kept the following characteristics which are found in many realistic applications:

a) A fleet of vehicles (possibly heterogeneous) housed at a central origin. There is no a priori restriction imposed over the size of the fleet.

b) A deterministic demand at each stop which must be entirely collected by the vehicle servicing the stop.

c) The generation of a single route for each vehicle, starting and ending at the origin.

d) The optional introduction of some of the following constraints:

   i) capacity constraints

   ii) maximal authorized distance (or time) traveled by each vehicle

   iii) time windows at each stop

e) The minimization of the operational costs which are defined in the user's formulae.

With this class of problems in mind, we designed a set of templates, known as the general heuristic, which supports various resolution strategies. This approach seems attractive for two main reasons:

a) The identification of a good resolution strategy, as mentioned before, is rarely obvious in face of problems for which no exact polynomially bounded algorithms exist. Hence, the concept of a general structure instantiated by the

user adds great flexibility to the process of resolution. It allows one to test a multiplicity of approaches simply by updating the appropriate formulae.

b) Since the general heuristic does not rely on a specific strategy, we are able to reproduce many classical algorithms described in the litterature. We should mention:

   i) insertion algorithms [8] , like:
      - nearest insertion
      - cheapest insertion
      - greatest angle insertion
      - etc...

   ii) nearest neighbor algorithm [2]

   iii) Clarke & Wright 's algorithm [3]

   iv) clustering algorithms
      - sweep algorithm of Gillett & Miller [5]
      - algorithm of Ferland & Rousseau for school bus routing [4]

Since the solving process is based on templates instantiated by the user, many extensions to the class of problems solved by ALTO can be looked upon with just slight modifications to the current system. These extensions include the introduction of multiple origins and the addition of new constraints involving the stops or the vehicles (like "exclusion constraints" - when some stops cannot be collected by the same vehicle - and multicommodity transportation).

The next section will now introduce the "object-oriented paradigm" used to design ALTO. This paradigm is very important since the general heuristic is modelized as a set of operators acting upon classes of objects.

Section 2. The object-oriented paradigm

An object-oriented programming language, called LOOPS [9, 10], was used to implement our system. We chose the object-oriented paradigm because any vehicle routing problem can be easily partitionned into some well defined entities such as a network with nodes and links, an origin, a set of stops and a fleet of vehicles. Hence, the organization of the program as a set of related objects stems naturally from the application. Moreover, since it is our intent to introduce in a near future some rule-based deduction mechanisms, this symbolic representation appears particularly attractive [1].

In section 2.1, the basic concepts regarding LOOPS and object-oriented programming are presented. Obviously, this section does not give a detailed description of the language, but rather points out some features of special interest for our application. Section 2.2 then presents the general organization of the system.

2.1 LOOPS: an object-oriented programming language

LOOPS, like most object-oriented programming languages, distinguish between two kinds of objects: classes and instances. A class is a description of one or more similar objects while an instance refers to a particular occurrence of a given class. Objects combine the properties of procedures and data: they store data in variables and respond to messages by carrying out procedures (methods).

The variables stored in the objects are divided in two categories: class variables (CV) which are used to hold information shared by all instances of a class and instance variables (IV) which contain information specific to a particular instance. Hence, each instance holds privately the values of its instance variables and the classes contain the class variables, methods and instance variable names, with possibly default values. The classes so defined are then combined to create multiple inheritance lattices (directed acyclic graphs) allowing an incremental specialization of their basic components. By this mean, any class inherits variables and methods from the classes defined higher in the lattice.

There are also special kinds of classes, called "mixins", which cannot give rise to particular occurrences. Such mixins are used primarily to add new properties, variables and methods to the basic classes.

The next section will now explain how ALTO uses these features to organize its classes of objects.

2.2 The classes of objects in ALTO

The inheritance lattice used by the system is shown in figure 2. Six subclasses of OBJECT, which holds properties shared by all objects of the system, are defined for our application: NETWORK, NODE, LINK, STREET, VEHICLE and ROUTE. The class NODE is then divided into the specialized classes SIMPLE-NODE and SPECIAL-NODE. Simple nodes are located where the links of the network intersect and in urban networks these nodes correspond to the corners of the streets. A special node, on the other hand, is either a STOP or an ORIGIN.

Some mixins (the unboxed classes in figure 2) are also included in the lattice in order to add some specific properties to the basic classes mentioned above. For example, NETWORK-DISPLAY contains all the variables and methods allowing the display of the network on

the screen, NETWORK-INTERFACE controls some specific input/output functions while NETWORK-DIJKSTRA, NETWORK-HEURISTIC and NETWORK-TS (for Traveling Salesman) deal with the effective implementation of the various routing algorithms.

Many other interrelations among the symbolic objects exist beside the hierarchical links shown in figure 2. For example, a network is in fact a composite object made of nodes and links. Hence, any instance of the class NETWORK contains two variables, NODE-LIST and LINK-LIST, designed to hold pointers to the associated NODE and LINK instances. In the same way, a street is a composite object made of links.

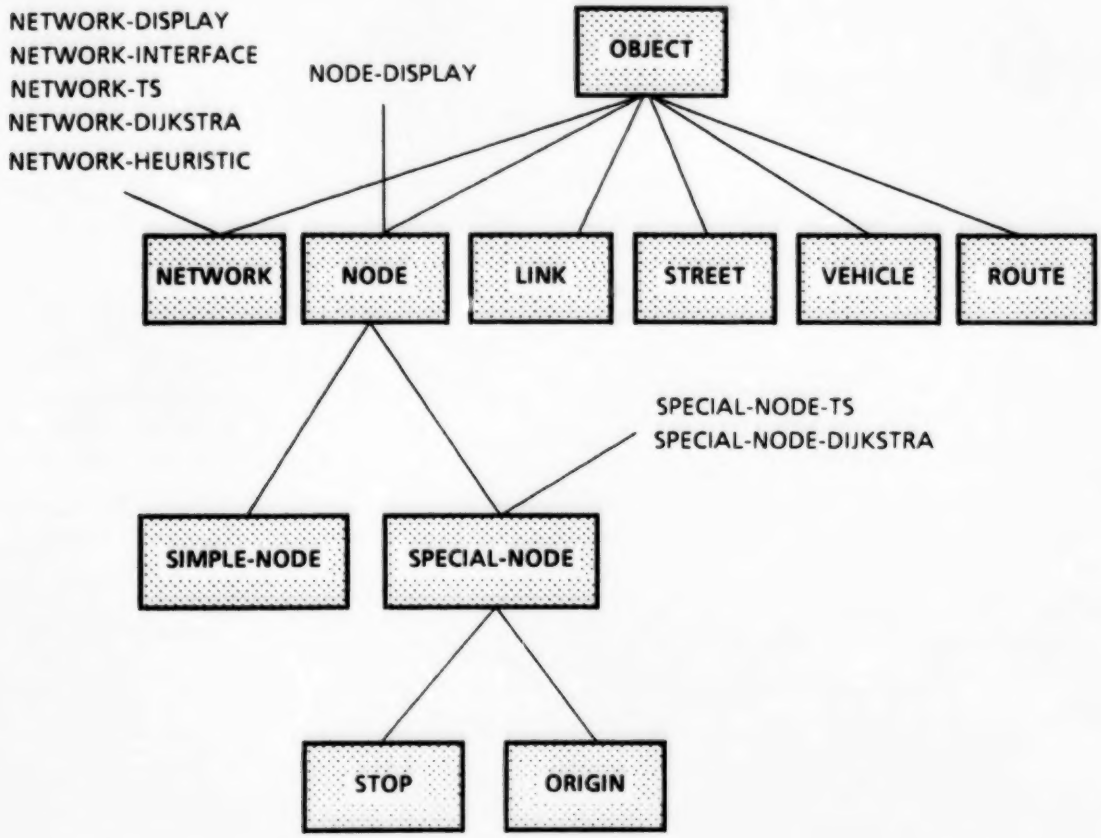

Fig. 2 The classes of objects in ALTO

Given the inheritance lattice shown in figure 2, we can now describe in more detail the class NODE and its subclasses in order to underline the benefits of object-oriented programming.

2.2.1 The class NODE and its related subclasses

Since our system can solve what is called node routing problems, the class NODE and its subclasses deserve a special attention. In figure 3, the node's hierarchy is detailed in order to include some illustrative variables and methods. This figure shows the incremental specialization of the various classes, as we go from a NODE to a STOP or an ORIGIN. The information stored in these classes is organized to avoid redundancy. Hence, the class NODE contains general attributes and methods common to all nodes in a network like an identifier, the coordinates, the list of incident links and the methods to manipulate these informations. The descendants of this class contain more specialized attributes concerning restricted subsets of nodes in the network. For example, instance variables like DEMAND and TIME WINDOW in the class STOP are not accessible to the simple nodes or the origin. In the same way, the origin is the only node concerned with the management of the fleet of vehicles.

As mentioned earlier, class variables contain values shared by all instances of a given class. In particular, the variable MENU in the class STOP contains a specialized menu, called MENU-STOP, allowing the user to modify interactively the content of any desired stop. Selections made in this menu will trigger the methods contained in the class, like MODIFY-DEMAND. Here, the method SHOW-MENU, created to display and manage MENU-STOP, is just a specialization of a more general method which is stored in the class NODE. Hence, the method SHOW-MENU in the class STOP handles exclusively specific selections concerning the stops and refers to the method SHOW-MENU in the class NODE for more general selections, like MODIFY-COORDINATES.

These observations show how the hierarchical organization of nodes allow a judicious partition of the attributes based on semantic considerations. In fact, a stop (or an origin) has a special meaning in a transportation network and the concept of specialization allows us to design classes that encapsulate variables and methods designed specifically for certain kinds of nodes.

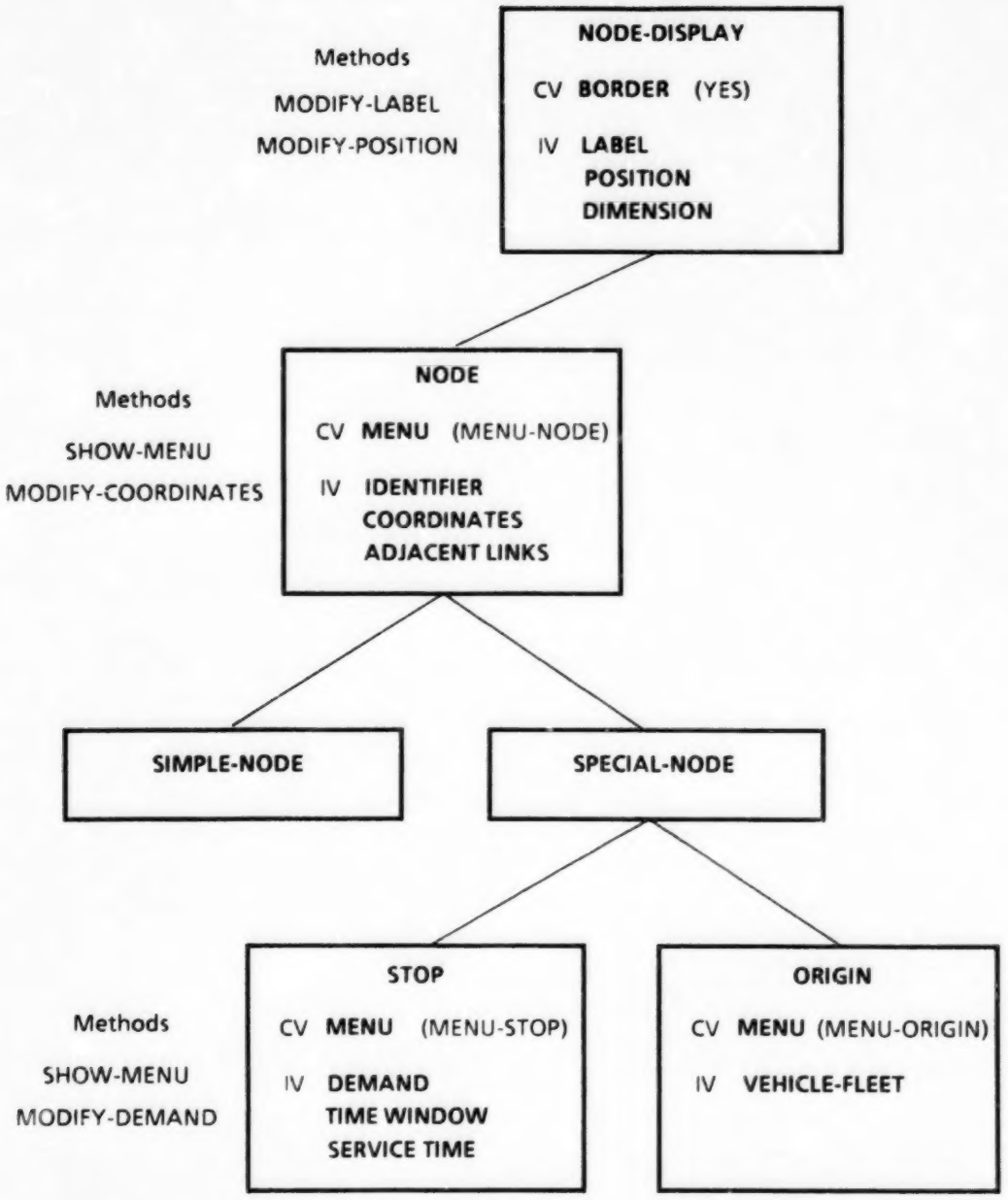

Fig. 3 The class NODE and its related classes

2.2.2 The benefits of object-oriented programming

Generally speaking, the object-oriented paradigm enhances the system in the following ways:

a) It is possible to define as many attributes as desired in a given class. For example, a vehicle is described in our system by a set of characteristics like speed, capacity and maximal distance or travel time. Obviously, this set can be increased to suit new applications with some specific requirements.

b) The concept of specialization in a multiple inheritance lattice allows the system to deal with different types of nodes in a given network. The specialization of nodes is a fundamental concept in transportation problems for which object-oriented programming is very well suited.

c) Default values are used to caracterize the occurrence of typical instances of a given class. These defaults are used in particular to simplify the creation of an homogeneous fleet of vehicles. If needed, the user can modify interactively these default values.

d) Mixins are useful to encapsulate variables and methods providing some well identified facilities (cf. the mixins related to the class NETWORK). In fact, it is a valuable tool to build large programs incrementally.

e) We can foresee natural extensions to other fields involving networks. For example, many parts of the program can be used, with just little modifications, to handle computer networks. To this mean, the class NODE must be specialized according to the nature of the nodes involved (i.e. Hosts, IMPs ...).

In addition, the INTERLISP-D environment allows the creation of "inspection windows" displaying the content of any object specified by the user (figure 4). This facility adds a "distributed data base" aspect to the system. It is a simple feature to implement since each object is an individual entity enclosing the relevant attributes. With such a facility, the user is able to monitor the status of any object involved in the definition of the problem such as the current length of the routes, the service time at a given stop, etc.

There is no doubt that the object-oriented paradigm adds generality and flexibility to the current system. In particular, it allows us to define a problem as a set of interrelated entities providing informations to the general heuristic during the solving proces, as it will now be explained.

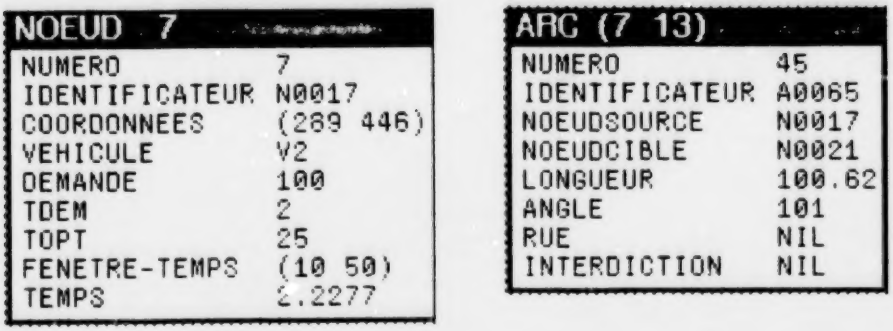

Fig. 4 Inspection windows for a node and a link

Section 3. The general heuristic

Fundamentally, the general heuristic is composed of four basic operators manipulating sets of objects (figure 5).

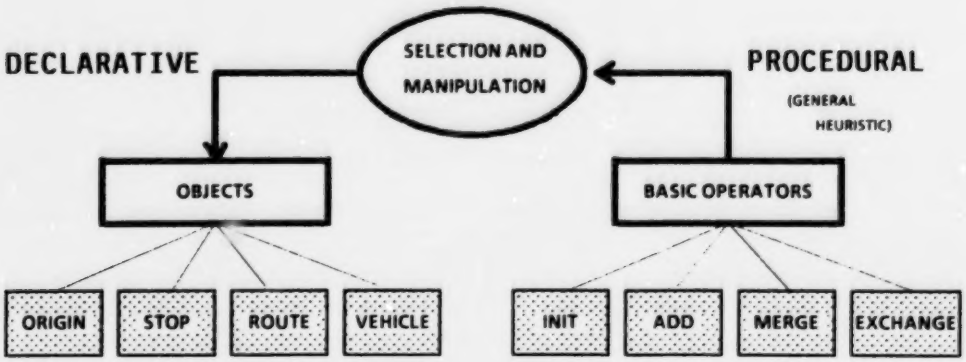

Fig. 5 Objects and basic operators

In the figure, we can identify three major elements:

a) Objects

The objects have attributes providing informations to the operators acting over them. In particular, the major attributes used by the general heuristic are:

i) for the origin : coordinates and number of vehicles available.

ii) for the stops : coordinates, time window and demand.

iii) for the routes : number of stops, length, travel time and total demand.

iv) for the vehicles : speed, capacity and maximal authorized distance or travel time.

b) Basic operators

The objects are manipulated by four basic operators symbolizing distinct phases of the general heuristic. These operators are:

i) INIT : operator for the creation and initialization of new routes.

ii) ADD : operator for the insertion of new stops into existing routes.

iii) MERGE : operator for a two-by-two merging of the routes

iv) EXCHANGE : operator for the modification of the sequence of stops in the routes.

The templates used to modelize each one of these four basic operators will be presented in sections 4,5,6 and 7.

c) A selection mechanism

Each operator first selects one or more objects and then manipulates these objects. The manipulation process is intrinsically defined in each operator but the selection process is put under user's control by means of selection formulae. These formulae associate a value to the objects involved in the problem's definition. Such values, called OMC for Object Measure Coefficient, allows ordering and selection of objects (for example, we can select the object with minimal or maximal OMC). The most usual selection formulae involve real functions of the following types:

i) $f : E_n \rightarrow \mathbb{R}$ (ex: the demand at a stop)

ii) $f : E_r \rightarrow \mathbb{R}$ (ex: the length of a route)

iii) $f : E_v \rightarrow \mathbb{R}$ (ex: the capacity of some vehicle)

iv) $f : E_n \times E_n \rightarrow \mathbb{R}$ (ex: the distance between two nodes)

v) $f : E_n \times E_r \rightarrow \mathbb{R}$ (ex: the distance between a node and a route)

where

E_n = the set of nodes in the network

E_r = the set of routes

E_v = the set of vehicles

We will now describe in turn each one of the four basic operators INIT, ADD, MERGE and EXCHANGE.

Section 4. Operator INIT

INIT is an operator allowing the creation of new routes. For each route, one or more seeds are selected in the set of "free" stops, that is stops not already serviced by a vehicle. The process of creation and initialization is fundamental since the operator ADD can only insert stops into existing routes. Moreover, the seeds provide a way to specify a general orientation for each route. Figure 6 shows the basic principle. among the free stops (white squares), n_2 and n_4 are used to initialize route R_x, n_5 and n_6 initializes R_y and n_7 initializes R_z.

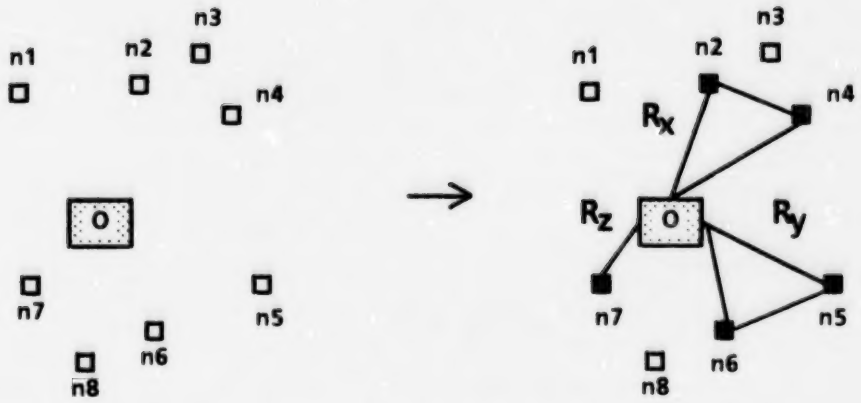

Fig. 6 Initialization of routes Rx, Ry and Rz

The template for operator INIT is presented in the following figure:

INIT

N ← **number of routes;**

SE_k ← **subset of free stops;**

While N > 0 and vehicles are still available
do

SEEDS ← **selection of seeds;**

v ← **selection of a vehicle;**

create a route where the SEEDS are serviced by vehicle v;

SE_k ← SE_k - SEEDS;

N ← N - 1;

Fig. 7 Operator INIT

The bold portions of the template are the decision points where the user can intervene. Hence, it is possible to specify the number of routes desired, the subset of free stops

over which the selection of seeds will proceed and finally the selection formulae for the seeds and the vehicle.

Initialization can also be executed "manually": we mean here that a user is able, via menus, to associate interactively individual free stops to available vehicles. This alternative seems attractive since expert users can often initialize the routes in a more judicious way than any other algorithmic approach.

Section 5. Operator ADD

ADD is an iterative procedure for the insertion of free stops into existing routes. At each iteration, two fundamental decisions must be made: a free stop has to be selected and then inserted at a proper location in a route. The principle is depicted in figure 8: here, the free stop n_3 is selected and inserted between n_2 and n_4 in route R_x. The procedure is then repeated for each of the remaining free stops.

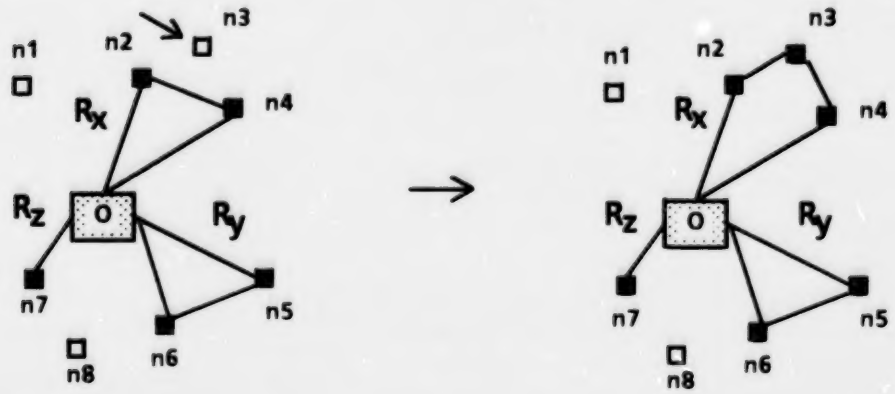

Fig. 8 Selection and insertion of stop n3

The template for operator ADD is presented in the following figure:

ADD

SE_k ← **subset of free stops;**

While $SE_k \neq \emptyset$

do

r_1 ← **selection of route 1;**

k ← **selection of a free stop ;**

r_2 ← **selection of route 2 ;**

i ← **selection of a stop in a route ;**

if **stopping criterion** verified then exit the loop and stop ;

if the insertion of k satisfies the constraints then

insert k **before / after** i

else

use **mode ;**

remove k from SE_k ;

Fig. 9 Operator ADD

As for INIT, the bold portions are decision points where the user can intervene. In the two following subsections we will give particular attention to the selection of routes and stops and to the specification of constraints.

5.1 Selection of routes and stops

As described in figure 9, a selection and insertion procedure is applied to the free stops included in SE_k. The procedure is repeated until either SE_k is empty or a stopping criterion, defined by the user, is verified. At each iteration, four different selection formulae are evaluated:

a) the selection of route 1 (r_1)

A particular route can be identified to guide the subsequent selection of the free stop k. For example, a user can select the shortest route in order to retain the nearest free stop from that route. Hence, it is possible to reproduce a sequence of events like: route selection, free stop selection and insertion of the free stop in the route. We must note however that the specification of r_1 is optional and the selection will not be activated if the associated formula is NIL.

b) the selection of a free stop (k)

This formula identifies a particular stop in SE_k. This formula is fundamental, because it defines the order in which the free stops are considered for inclusion into existing routes.

c) the selection of route 2 (r_2)

A second route can be identified after the selection of the free stop. For example, a user can select a free stop and identify the nearest route from that free stop. Hence, it is possible to reproduce a sequence of events like: free stop selection, route selection and insertion of the free stop in the route. As for r_1, the selection of r_2 is optional.

d) the selection of a stop in a route (i)

Here, the selection formula identifies a stop already included in a route. If a particular route has been selected, the selection formula for i will only be applied to the stops in that route. Otherwise, all the routes are considered. The identification of i along with an insertion order (that is, before or after i) defines a unique location for the insertion of k.

If desired, the step-by-step execution of this iterative procedure can be visualized by the user. To this mean, each selected free stop blinks on the screen before its insertion and the routes are updated and displayed accordingly. Such a facility provides a way to analyze closely the behavior of the operator and detect flaws in the instantiation formulae.

5.2 The specification of constraints

Actually, four types of constraints can be introduced in a given problem: capacity constraints, vehicle's maximal autonomy in terms of distance or travel time and time windows. For each type, there are ponderation parameters for the adjustment of the constraints. These parameters are:

a) LAMBDAcap: for each route,

TOTAL DEMAND ≤ LAMBDAcap x VEHICLE'S CAPACITY

b) LAMBDAdist: for each route,

LENGTH ≤ LAMBDAdist x VEHICLE'S MAXIMAL AUTHORIZED DISTANCE

c) LAMBDAtime: for each route,

$$\text{TRAVEL. TIME} \leq \text{LAMBDAtime} \times \text{VEHICLE'S MAXIMAL AUTHORIZED TRAVEL TIME}$$

d) LAMBDAtw1, LAMBDAtw2: for each stop,

$$\text{SERVICE TIME} \leq \text{TOPT} + \text{LAMBDAtw2} \times (\text{TMAX} - \text{TOPT})$$

$$\text{SERVICE TIME} \geq \text{TOPT} - \text{LAMBDAtw1} \times (\text{TOPT} - \text{TMIN})$$

where TOPT = optimal service time
TMIN = lower bound of the time window
TMAX = upper bound of the time window

A value greater than 1 relaxes the associated constraint while a value smaller than 1 has the opposite effect. These ponderation parameters are very important since slight violation of the constraints can be tolerated if a solution has many other attractive features. Hence, it is an asset to be able to adjust the value of the parameters for a given problem.

Obviously, the constraints can prohibit the insertion of the free stop k in a given route. In such a case, five different modes are offered to handle k:

a) free mode: stop k is not inserted at the specified location and remains free.

b) alternative mode: operator ADD uses an alternative selection formula for i (defined by the user) which considers only locations where the insertion of k is possible. If no such location is found, stop k remains free.

c) insertion mode: free stop k is inserted in spite of the violation of the constraints.

d) interruption mode: operator ADD stops as soon as a free stop cannot be inserted at the specified location. Hence, all the remaining stops in SE_k stay free.

e) interactive mode: here, the user specifies interactively one of the four preceding modes whenever the insertion of a free stop violates some constraints. The interactive mechanism provides a way to handle the insertion of each free stop on an individual basis.

All these modes provide a refined way to handle insertion in constrained problems. For example, modes (a) and (d) allow the user to relax the constraints or modify some formulae in order to handle insertion of the stops that are left free after the first activation of operator ADD.

Section 6. Operator MERGE

MERGE is an iterative procedure allowing the creation of larger routes from smaller ones. The basic principle is depicted in figure 10: here, routes R_x and R_y are selected and merged to create route R_{xy}. The procedure is then repeated with the new set of routes.

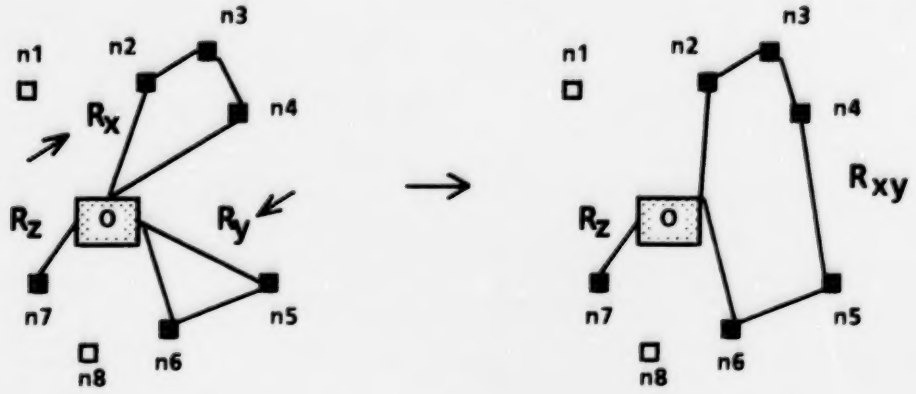

Fig. 10 Creation of route Rxy from Rx and Ry

Actually, the merging process simply links the routes from their both ends, thus producing four larger routes from which the one with minimal length is chosen (figure 11). Eventually, a more refined way to handle this process will be devised in order to allow the integration of user defined formulae.

$$\text{Route}_1 = (n_1, \ldots, n_m)$$
$$\text{Route}_2 = (n_1', \ldots, n_m')$$

$$\text{MERGE}_1 = (n_1, \ldots, n_m, n_1', \ldots, n_m')$$
$$\text{MERGE}_2 = (n_1, \ldots, n_m, n_m', \ldots, n_1')$$
$$\text{MERGE}_3 = (n_m, \ldots, n_1, n_1', \ldots, n_m')$$
$$\text{MERGE}_4 = (n_m, \ldots, n_1, n_m', \ldots, n_1')$$

Fig. 11 Four ways to merge two routes

The template for operator MERGE is presented in the following figure:

MERGE

SE_r ← **subset of routes** ;

While $|SE_r| > 1$

do

r_1 ← **selection of route 1** ;

remove r_1 from SE_r ;

r_2 ← **selection of route 2** ;

remove r_2 from SE_r ;

if **stopping criterion** verified then exit the loop and stop ;

if merging r_1 and r_2 satisfies the constraints then

v ← **selection of the vehicle servicing r_1 or r_2** ;

create a new route r serviced by vehicle v

by merging r_1 and r_2 ;

add r to SE_r

else

use **mode** ;

Fig. 12 Operator MERGE

Here, a merging procedure is applied iteratively to the routes included in SE_r. This pocedure ends when the cardinality of SE_r drops under two or when a stopping criterion, defined by the user, is verified. At each iteration, three different selection formulae are evaluated:

a) the selection of route 1 and route 2 (r_1, r_2)

The selection formulae for r_1 and r_2 are fundamental since they define the order in which the routes are merged. Note that the new route r, obtained from r_1 and r_2, is added to SE_r and can be merged again with other routes.

b) the selection of the vehicle (v)

If the two vehicles servicing r_1 and r_2 are suitable for the new route r, the selection formula for the vehicle will select one of them. Obviously, if only one vehicle can service the route, this formula won't be evaluated.

As for operator ADD, the constraints can prohibit the merging process for two given routes. In such a case, four different modes are offered to handle r_1 and r_2:

a) free mode: routes r_1 and r_2 are not merged.

b) merge mode: routes r_1 and r_2 are merged in spite of the violation of the constraints.

d) interruption mode: operator MERGE stops as soon as the two selected routes r_1 and r_2 cannot be merged.

e) interactive mode: here, the user specifies interactively one of the three preceding modes whenever the merging process violates some constraints.

All these modes can be used to control more closely the merging process in constrained problems.

Section 7. Operator EXCHANGE

EXCHANGE modifies the sequence of stops in existing route allowing intra-routes as well as inter-routes exchanges. In this way, the initial assignment of stops to vehicles can be modified in a global optimization perspective. We use the fact here, that any Multiple Traveling Salesmen Problem (MTSP) can be converted into an equivalent Traveling Salesman Problem (TSP), by creating M copies of the origin and M copies of its incident links [1]. Using this transformation, the set of all existing routes can be coalesced into a single "super-route" which is then suitable for a 2-opt exchange procedure (figure 13).

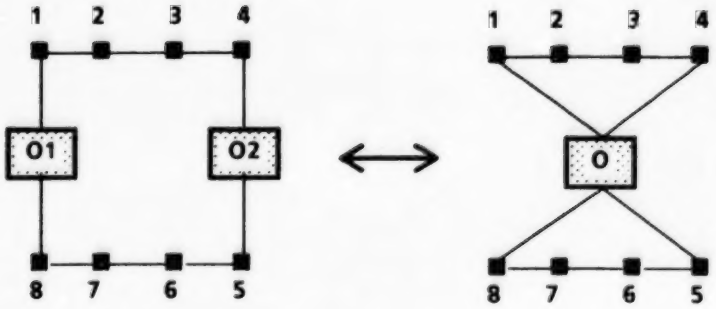

Fig. 13 Equivalence TSP - 2TPS

Taking advantage of the multiple origins found on the super-route, we extended the classical 2-opt exchange procedure by considering exchanges leading to the partition of the super-route into smaller subroutes (figure 14b). Such a partition leads to a feasible solution iff at least one copy of the origin is included in each subroute. These subroutes can then be recombined into an equivalent single route, simply by exchanging "equivalent edges", that is the edges connecting a given node to each copy of the origin (figure 14 c).

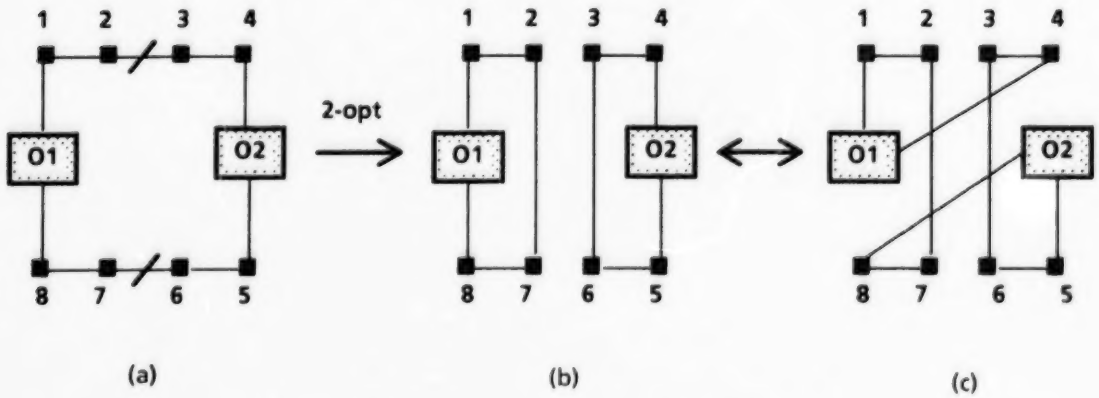

Fig. 14 An extended 2-opt exchange procedure

The template for operator EXCHANGE is described in the following figure:

EXCHANGE

```
QUALITYmax ← 0;
CURRENT SOLUTION ← initial solution;
While exchanges have not been tried in CURRENT SOLUTION
do
        EXCHANGE SOLUTION ← solution obtained after exchange;
        if EXCHANGE SOLUTION satisfies the constraints then
            QUALITY ← evaluation of the quality formula
                                    for EXCHANGE SOLUTION;
            if QUALITY > QUALITYmax then
                QUALITYmax ← QUALITY;
                CURRENT SOLUTION ← EXCHANGE SOLUTION;
```

Fig. 15 Operator EXCHANGE

The user can intervene in the definition of the quality formula. To this mean, six different criteria can be introduced:

c_1) length of the routes

c_2) compacity of the routes (as defined in [4])

c_3) saturation of the vehicle's capacity

c_4) saturation of the vehicle's maximal authorized distance

c_5) saturation of the vehicle's maximal authorized travel time

c_6) service time at the stops

The creation of the quality formula takes into account recent works in multicriteria decision analysis [6,7]. Fundamentally, the technique creates a quality formula by aggregating all the criteria into a single global utility function. To this mean, partial utility functions are created for each criterion involved in the formula and the value returned by the global function is the sum of the values of the partial utility functions.

For our application, we devised symmetric partial utility functions like the one shown in figure 16. As depicted in the figure, an optimal interval is defined and the criterion's utility decreases quadratically on both sides of that interval. Obviously, the value returned by the

partial utility function cannot be arbitrarily low since the associated criterion would otherwise take an overwhelming importance in the quality formula. Hence, an admissible interval is defined so that the utility function reaches 0 at the bounds of that interval. For example, if we consider the quality of a solution in regard to the saturation of the vehicle's capacity, the admissible interval can be set to [0%, 100%] and the optimal interval to [40%, 60%] in order to avoid overutilization (> 60%) as well as underutilization (< 40%) of the vehicles.

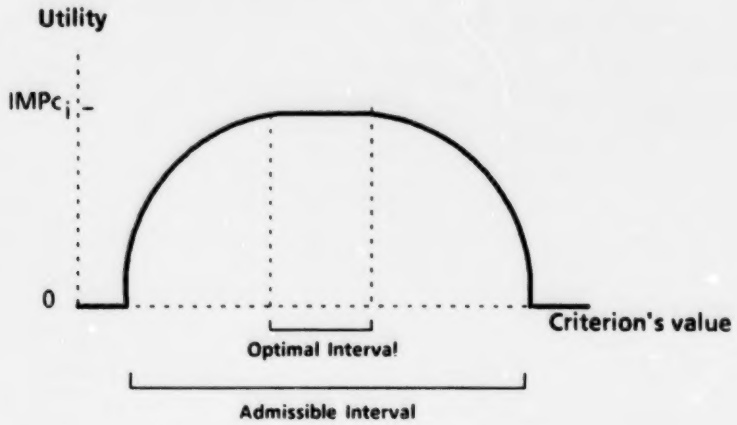

Fig. 16 Utility function for a criterion c_i

In this regard, the parameters needed to define a partial utility function for a criterion c_i are the followings:

a) a set of objects (E_{c_i})

The criterion c_i is applied to each object in the set E_{c_i}. For the six criteria introduced previously, these sets are:

i) for c_1 and c_2: the set of routes.

ii) for c_3, c_4 and c_5: the set of vehicles.

iii) for c_6: the set of stops.

b) the weight of the criterion ($WEIGHT_{c_i}$)

The weight is used to evaluate the importance of c_i in the multicriteria quality formula. More precisely, the importance of the criterion c_i is:

$$IMP_{c_i} = \frac{WEIGHT_{c_i}}{\sum_{j=1}^{m} WEIGHT_{c_j}}$$

where m = number of criteria in the quality formula

c) an optimal interval $[OPT1_{c_i}, OPT2_{c_i}]$

This interval defines the set of values for which the quality of the solution is maximal in regard to the criterion c_i. If this optimum is a single value then: $OPT1_{c_i} = OPT2_{c_i}$.

d) an admissible interval $[ADM1_{c_i}, ADM2_{c_i}]$

This interval is used to bound the partial utility function. Actually, it is created by specifying a maximal allowed gap ($MAXGAP_{c_i}$) between the optimal interval and the criterion's value. If this gap equals 0 then the value is in the optimal interval and the criterion's utility is maximal, that is equals IMP_{c_i}. On the other hand, the criterion's utility is minimal and equals 0 if the gap is greater than $MAXGAP_{c_i}$.

e) Two multipliers ($MULT1_{c_i}$ and $MULT2_{c_i}$)

If the value of the criterion c_i, when applied to an object e in the set E_{c_i}, is less than $OPT1_{c_i}$, then the gap between this value and the optimal interval is:

$$GAP_{c_i e} = MULT1_{c_i} \times (OPT1_{c_i} - VALUE_{c_i e}).$$

If the value of the criterion ci, when applied to an object e is greater than $OPT2_{c_i}$, then the gap is:

$$GAP_{c_i e} = MULT2_{c_i} \times (VALUE_{c_i e} - OPT2_{c_i})$$

These multipliers can be used to penalize more or less the difference between the criterion's value and the optimal interval. It can be useful if this difference has not the same meaning on both sides of the optimal interval. In such a case $MULT1_{c_i} \neq MULT2_{c_i}$.

With all these parameters, a partial utility function f_{c_i} can be created. This function returns the utility of the criterion c_i and is defined as follows:

$$f_{c_i}(x_{c_i}) = \begin{cases} IMP_{c_i} \times (1 - x_{c_i}^2) & \text{if } x_{c_i} \in [0, 1] \\ 0 & \text{otherwise} \end{cases}$$

where

$$x_{c_i} = \frac{1}{|E_{c_i}|} \sum_{e \in E_{c_i}} \frac{GAP_{c_i e}}{MAXGAP_{c_i}}$$

$$GAP_{c_i e} = \begin{cases} MULT1_{c_i} \times (OPT1_{c_i} - VALUE_{c_i e}) & \text{if } VALUE_{c_i e} < OPT1_{c_i} \\ MULT2_{c_i} \times (VALUE_{c_i e} - OPT2_{c_i}) & \text{if } VALUE_{c_i e} > OPT2_{c_i} \\ 0 & \text{otherwise} \end{cases}$$

As we can see, f_{c_i} always returns a value in the interval [0, IMPci]. Since the partial utility functions are additive, we get :

$$0 \leq f(x_{c_1}, \ldots, x_{c_m}) = \sum_{i=1}^{m} f_{c_i}(x_{c_i}) \leq \sum_{i=1}^{m} IMP_{c_i} = 1$$

where m = number of criteria in the quality formula.

Hence, the value returned by the global utility function is always in the interval [0, 1] and this value increases when the quality of the solution increases.

Having described the four basic operators, we can now conclude this section by emphasizing the fact that these operators are independent from each other and can be activated by the user in any desired sequence. The only restriction is obvious: the first operator of the sequence must be INIT (figure 17). Actually, the user must apply the operators manually but we can foresee in a near future automatic generation of sequences of properly instantiated operators (that is, automatic generation of programs!).

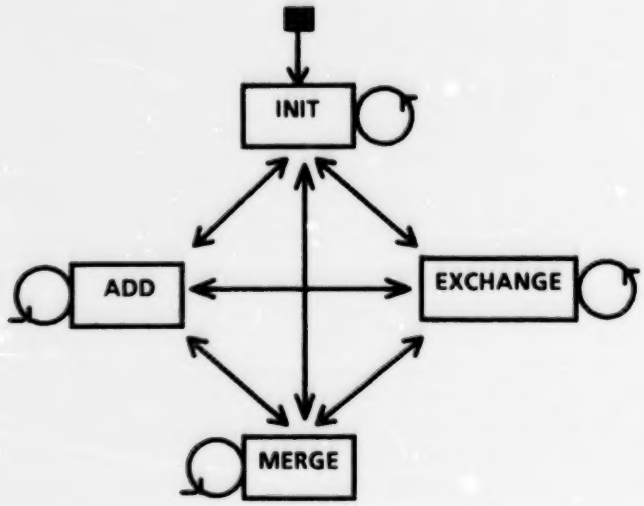

Fig. 17 Activation sequences for the basic operators

Section 8. Some practical considerations

The concept of a general heuristic implies a close interaction with the user. Therefore, a number of facilities are offered in order to simplify the process of creating formulae. In the following, we categorize these facilities and give for each of them a few examples.

a) Global variables

Various global variables are available in the system like the origin o, the total number of stops N_t, the total number of free stops N_k, the ponderation parameters LAMBDAcap, LAMBDAdist, LAMBDAtime, LAMBDAtw1, LAMBDAtw2, etc. These variables can be included at any time in the selection formulae.

b) Predefined sets

Some standard sets are already predefined like the set of routes E_r, the set of vehicles E_v, the set of seeds E_s, the set of free stops E_k and the set of stops included in the routes E_i. The interpretation of the latter set E_i can however be modified in operator ADD by the selection of a particular route (either r_1 or r_2). in such a case, E_i is interpreted as the set of stops in the selected route. Likewise, the interpretation of E_i is modified in operator MERGE by the specification of a restricted subset of routes SE_r.

Naturally, all the predefined sets are automatically updated by the system. For instance, any free stop inserted in a route is removed from E_k.

c) Set operators

The set operators provide a way to manipulate easily the predefined sets mentionned in (b). For example, (MAXOBJ SET FUNCTION) returns the object in SET for which the value of FUNCTION is maximal while (MAXVAL SET FUNCTION) returns the maximal value of FUNCTION. The operators MINOBJ and MINVAL have opposite interpretations. Finally, (SUBSET SET PREDICATE) returns the subset of objects verifying the predicate.

These set operators can be nested in order to create more complex expressions like (MINOBJ SET_1 (MAXVAL SET_2 FUNCTION)). In this example, MINOBJ returns the object in SET_1 for which the value of (MAXVAL SET_2 FUNCTION) is minimal.

d) Primitives

These primitives are predefined functions which can be incorporated into user's formulae. There are many such primitives in the system, so we will present here just a few of them (these primitives are used in the complete examples given at the end of the section).

- $ADD?_{ik}$: is it possible to add k in the route servicing i without violating the constraints?
- $ANGLEC_{ikj}$: "clockwise angle" (figure 18)
- $DIST_{ik}$: length of the shortest path from node i to node k
- $EMPTY?_E$: is set E empty?
- EM_{ik} : extra-mileage, i.e. $DIST_{ik} + DIST_{kj} - DIST_{ij}$

- $FIRST_r$: first stop on route r
- $LAST_r$: last stop on route r
- $MERGE?_{rr'}$: is is possible to merge routes r and r' without violating the constraints?
- NEW_{E_r} : route created by the last application of operator INIT
- $SAVING_{ik}$: "saving", i.e. $DIST_{io} + DIST_{ok} - DIST_{ik}$ (o is the origin)
- VEH_r : vehicle servicing route r

d) Usual mathematical and logical operators like +, -, x, /, AND, OR, NOT.

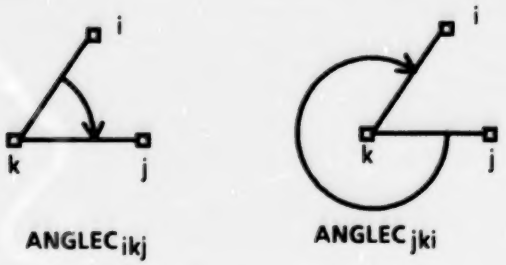

Fig. 18 Clockwise angle

With such facilities, the user can create selection formulae and use them to instantiate the general heuristic. In fact, we will show how to use these tools in order to reproduce three classical algorithms: the farthest insertion [8], the sweep algorithm of Gillett and Miller [4] and the Clarke and Wright's algorithm [3]. To this mean, the templates for operators INIT, ADD and MERGE will be properly instantiated. In these three examples, the fleet is homogeneous, o is the origin, r' and r'' $\in SE_r$, k' $\in SE_k$, s' $\in E_s$, i' $\in E_i$ and j' is the successor of i' (i' being in a route).

We must again emphasize the fact that these templates provide a general structure for the specification of user defined resolution strategies, and so are not really computationnally efficient. In fact, any given instantiation can introduce some redundant computations.

Example 1. Farthest insertion

INIT

N ← 1;
SE_k ← E_k;

While N > 0 and vehicles are still available
do
SEEDS ← **(MAXOBJ SE_k $DIST_{ok'}$)**;

(i.e. $OMC_{k'} = DIST_{ok'}$ and we select stop k' with maximal OMC)

v ← **(RANDOBJ E_v)**;

(i.e. random selection of vehicle v)

create a route where the SEEDS are serviced by vehicle v;

SE_k ← SE_k - SEEDS;
N ← N - 1;

ADD

SE_k ← E_k;

While $SE_k \neq \emptyset$
do
r_1 ← **NIL**;

k ← **(MAXOBJ SE_k (MINVAL E_i $DIST_{i'k'}$))**;

(i.e. $OMC_{k'}$ = (MINVAL E_i $DIST_{i'k'}$) and we select stop k' with maximal OMC)

r_2 ← **NIL**;

i ← **(MINOBJ E_i $EM_{i'k_j}$)**;

(i.e. $OMC_{i'} = EM_{i'k_j}$ and we select stop i' with minimal OMC)

if **NIL** verified then exit the loop and stop;

if the insertion of k satisfies the constraints then
insert k **after** i
else
use **free mode** ;

remove k from SE_k;

Constraints : no constraints

Example 2. Sweep algorithm

Here, the user applies iteratively operators INIT and ADD until no more free stops remain.

INIT

N ← 1;
SE_k ← E_k;

While N > 0 and vehicles are still available
do
SEEDS ← (if EMPTY?$_{E_r}$ then (MAXOBJ SE_k $DIST_{ok'}$)
else (MINOBJ SE_k (MINVAL E_s $ANGLEC_{s'ok'}$));

v ← (RANDOBJ E_v);

create a route where the SEEDS are serviced by vehicle v;

SE_k ← SE_k - SEEDS;
N ← N - 1;

ADD

SE_k ← E_k;

While SE_k ≠ Ø
do
r_1 ← NEW$_{E_r}$;

k ← (MINOBJ SE_k $ANGLEC_{LAST_{r_1}ok'}$);

r_2 ← NIL;

i ← $LAST_{r_1}$;

if NIL verified then exit the loop and stop;

if the insertion of k satisfies the constraints then
insert k after i
else
use interruption mode ;

remove k from SE_k;

Constraints : LAMBDAcap = 1.0

Example 3. Clarke and Wright's algorithm

INIT

```
N      ← Nt;
SEk    ← Ek;
While N > 0 and vehicles are still available do
      SEEDS   ← (RANDOBJ SEk);
      v       ← (RANDOBJ Ev);
      create a route where the SEEDS are serviced by vehicle v;
      SEk     ← SEk - SEEDS;
      N       ← N - 1;
```

MERGE

```
SEr ← Er;
While |SEr| > 1 do
      r1 ← (MAXOBJ SEr                          (* r' iterates over SEr in MAXOBJ while
            (MAXVAL SEr - {r'}                     r'' iterates over SEr - {r'} in MAXVAL *)
                 (if MERGE?r'r'' then
                       MAX { SAVINGFIRSTr'·FIRSTr'' , SAVINGFIRSTr'·LASTr''
                             SAVINGLASTr'·FIRSTr'' , SAVINGLASTr'·LASTr''}
                       else -∝ )));
      remove r1 from SEr;
      r2 ← (MAXOBJ SEr (if MERGE?r1r'
                           then MAX { SAVINGFIRSTr1FIRSTr' , SAVINGFIRSTr1LASTr'
                                      SAVINGLASTr1FIRSTr' , SAVINGLASTr1LASTr'}
                           else -∝ ));
      remove r2 from SEr;
      if NIL verified then exit the loop and stop;
      if merging r1 and r2 satisfies the constraints then
            v ← VEHr1;
            create a new route r serviced by vehicle v by merging r1 and r2;
            add r to SEr
      else
            use interruption mode ;
```

Constraints : LAMBDAcap = 1.0

As mentionned earlier, it is possible to reproduce many other classical algorithms with our heuristic. Moreover, since the templates are instantiated with user defined formulae, an expert can design new resolution approaches in face of complex problems. In fact, the next section presents such an application for a real mail pick-up problem.

Section 9. A real application

In order to illustrate the potential benefits of ALTO, we applied it to a real mail pick-up problem. The network displayed in figure 19 is the BOURASSA postal zone located in the north of the city of Montreal. Each stop here is a post office which must be serviced within a fixed time span. For this application, the violation of the lower bound of the time window is tolerated and implies only an additional waiting time for the vehicle (up to the lower bound, when the mail is available). For each vehicle, the speed is set at 35 kilometers per hour and the maximal authorized travel time, which must not be exceeded, is set at 120 minutes. Finally, there is no a priori restriction on the number of vehicles to be used.

With templates such as those offered by the general heuristic, we can solve this problem in many different ways, trying new or already known resolution strategies. For example, we applied strategies allowing relaxation of the time windows during the routes generation, using the exchange operator to attain feasibility. Other strategies were devised in order to insure the feasibility of the solution during the generation phase. Obviously, many different formulae were used to instantiate the templates and the approach now presented is the one that produces the minimal length solution. This approach is in fact an adaptation of the clustering technique designed by Ferland & Rousseau [4] which was modified to take care of the temporal constraints.

Here, the clusters are generated sequentially, each cluster being initialized with a seed which is the farthest free stop from the origin. The distance between the remaining free stops and the previously selected seed is then evaluated, using the formula of Ferland & Rousseau. The closest free stop is then selected and inserted at the minimal extra-mileage location which satisfies the upper bound of the time window for all the stops already included in the route. If there is no such location, the selected stop remains free and has to be serviced later by some other vehicle. Our approach is an adaptation of the clustering technique of Ferland & Rousseau in the sense that we use the constraints to build feasible routes (as opposed to "real" clusters where the stops are inserted at random). Such a stategy can be easily reproduced with a proper instantiation of the operators INIT and ADD, as illustrated in figure 20.

The procedure is then repeated in order to create enough routes to cover all the stops. The main benefit gained with this approach stems from the fact that the routes are built along spatial and temporal dimensions. In fact, all the routes generated in this manner are feasible and satisfies the upper bound of the time window at each stop. Moreover, since the insertion

process responds to the minimal extra-mileage criterion, the routes are already of reduced length, even before the application of any exchange strategy. This approach coupled with a 2-opt exchange procedure allowed a ten per cent reduction in the length of the routes as compared with a solution produced by a specialized algorithm called POST9. This last algorithm, developped at the CRT (Centre de Recherche sur les Transports, University of Montreal), was already superior to expert humans building the routes manually.

In figure 21, we present the routes produced by POST9 and by our algorithm. For each route, we show the total distance, the travel time and the sequence of stops. As we can see, the POST9 solution is 49.8 kms long while the ALTO solution is only 44.3 kms long. The superiority of our system stems from the fact that we can modify at will the resolution strategies in order to adapt them to the problem to be solved. Here, the experimentation work suggests that an explicit consideration of the time window during the clustering phase has positive effects on the final solution. This example shows how the flexibility introduced in the solving process of ALTO can be used to devise new heuristic strategies in face of complex routing problems for which no exact algorithms exist.

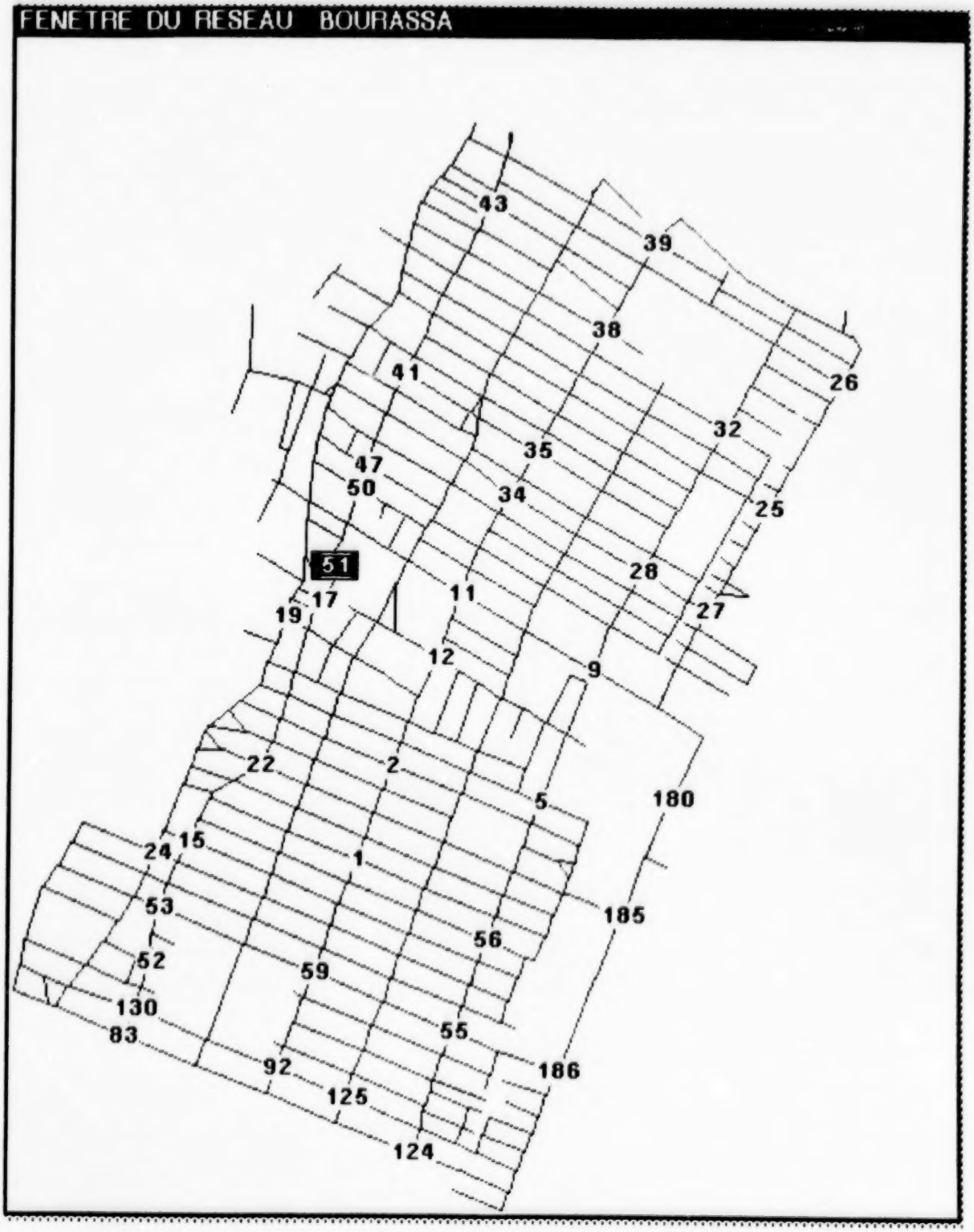

Fig. 19 The Bourassa postal zone with numbered mail offices and depot 51

Here, the user applies iteratively operators INIT and ADD until no more free stops remain.

INIT

N ← 1;
SE_k ← E_k;
While N > 0 and vehicles are still available do

SEEDS ← **(MAXOBJ** SE_k $DIST_{k'o}$**)**;
v ← **(RANDOBJ** E_v**)** ;
create a route where the SEEDS are serviced by vehicle v;
SE_k ← SE_k - SEEDS;
N ← N -1;

ADD

SE_k ← E_k;
While SE_k ≠ ∅ do

r_1 ← **NEW**$_{E_r}$;
k ← **(MINOBJ** SE_k (formula of Ferland & Rousseau with LAMBDA = 1.0)
(MINVAL E_s
(if $DIST_{s'k'} \leq DIST_{s'o}$ **then**

$$DIST_{s'k'}^2 \times (1 + LAMBDA \times (1 - \frac{DIST_{s'k'}^2 - DIST_{k'o}^2 + DIST_{s'o}^2}{2 \times DIST_{s'k'} \times DIST_{s'o}}))$$

else $DIST_{s'k'}^2 \times (1 + LAMBDA \times (DIST_{k'o} / DIST_{s'k'})))))$;
r_2 ← **NIL**;
i ← **(MINOBJ (SUBSET** E_i **ADD?**$_{i'k}$**)** $EM_{i'kj'}$**)**;
if **NIL** verified then exit the loop and stop;
if the insertion of k satisfies the constraints then
insert k **after** i
else
use **free mode** ;
remove k from SE_k.

Constraints : LAMBDAtw2 = 1.0 LAMBDAtime = 1.0

Fig. 20 Algorithm designed with ALTO

POST9	SEQUENCE OF STOPS	LENGTH (kms)	TRAVEL TIME (min)
ROUTE1	(53 59 56 125 124 92 83 52 15 1 55 186 130)	17.3	78
ROUTE2	(180 25 28 34 35 39 26 32 9 185 2)	14.5	69
ROUTE3	(11 12 19 47 41 43 50 22 24 17)	11.2	61
ROUTE4	(38 27 5)	6.8	53
TOTAL		**49.8**	**261**

ALTO	SEQUENCE OF STOPS	LENGTH (kms)	TRAVEL TIME (min)
ROUTE1	(53 56 59 92 124 125 55 83 52 15 17)	12.0	68
ROUTE2	(180 25 34 50 43 39 32 9 185 2)	14.5	65
ROUTE3	(12 38 26 27 28 35 41 47)	7.4	70
ROUTE4	(19 11 5 186 1 24 130 22)	10.4	50
TOTAL		**44.3**	**253**

Fig. 21 ALTO versus POST9

Section 10. Toward an Expert System for routing and scheduling problems

Section 9 showed that ALTO is a very useful system as such, but it is also viewed as a first step toward the generation of an expert system for routing and scheduling problems. The ultimate goal is to incorporate into the system the expertise needed to design "good" heuristic algorithms. In fact, this knowledge would allow automatic instantiation of the general heuristic in regard to the main characteristics of the problem to be solved. The major steps leading to such an expert system are the followings:

a) The conception of a general classification for routing and scheduling problems

The problems must first be classified according to some well defined criteria like the characteristics of the stops (number of stops, small / large demand...), the characteristics of the fleet of vehicles (number of vehicles, homogeneous or heterogeneous fleet...), the constraints involved and the specification of the operational costs.

b) The identification of the main characteristics of a given problem

Using the classification mentionned in (a), the next task is to identify the problem's characteristics which could influence the choice of the resolution strategy. To this mean, ALTO looks as a very useful tool for the experimentation of various resolution strategies in face of various kinds of problems, thus enhancing the development of the expertise.

c) The automatic deduction of good resolution strategies

The expertise gained in steps (a) and (b) must then be integrated into the system in order to provide autonomous deduction mechanisms for the automatic instantiation of the general heuristic. The proper instantiation will be deduced from the characteristics of the problem to be solved, using the knowledge embedded in the system.

d) The integration of interactive facilities allowing "manual" modifications

Since a computer program cannot take into account all the specific requirements of a particular application, it is essential to provide interactive facilities allowing manual modifications to the routes produced by the system. In this way, the user can always adapt the solution to his own needs.

Such an expert system would naturally be of great help for people designing heuristic algorithms for routing and scheduling problems, since it would guide them directly toward the best strategies.

Conclusion

In conclusion, ALTO is a sophisticated tool for the design and experimentation of routing algorithms. The major contribution of this system is twofold. First, it offers a rich environment supporting a dynamic solving process where a solution is built incrementally by the user with the help of graphic and interactive facilities. Finally, it incorporates a general heuristic which unifies a broad class of routing algorithms. Since this general heuristic corresponds to a set of templates instantiated by the user, it allows the design and experimentation of new heuristic algorithms in face of complex routing problems for which no exact polynomially bounded algorithms exist.

References

[1] Aikins J. S., "Prototypical Knowledge for Expert Systems", Artificial Intelligence 20, pp. 163-210 (1983).

[2] Bodin L., Golden B., Assad A. & Ball M., "Routing and Scheduling of Vehicles and Crews: The State of the Art", Computer and Operations Research, vol. 10, no. 2 (1983).

[3] Clarke G. & Wright J.W., "Scheduling of Vehicles from a Central Depot to a number of Delivery Points", Operations Research 12 , pp. 568-581 (1964).

[4] Ferland J. A., Rousseau J.M. & Chapleau L., "Clustering for routing in densely populated areas", European Journal of Operations Research 20, pp. 48-57 (1985).

[5] Gillett B. & Miller L., "A Heuristic Algorithm for the Vehicle Dispatch Problem", Operations Research 22, pp. 340-349 (1974).

[6] Jacquet-Lagreze E., "PREFCALC: Evaluation et Décision Multicritères", manuel de l'usager, EURO-DECISION eds. (1983).

[7] Potvin J.Y., "Un Système Informatique pour le Développement et l'Expérimentation d'Algorithmes de Génération de Tournées", Thèse de Doctorat, Université de Montréal (1987).

[8] Rosenkrantz D., Sterns R. & Lewis P., "An Analysis of several Heuristics for the Traveling Salesman Problem", SIAM J. Comp. 6, pp.563-581 (1977).

[9] Stefik M., Bobrow D., Mittal S. & Conway L., "Knowledge Programming in LOOPS", The AI Magazine, Fall 1983, pp. 3-13.

[10] Stefik M. & Bobrow D., "Object-Oriented Programming: Themes and Variations", The AI Magazine, Winter 1986, pp. 40-62.

[11] Teitelman W. & Masinter L., "The Interlisp Programming Environment", Computer, April 1981, pp. 25-32.

[12] Xerox Corp., Interlisp-D Reference Manual (1986).